NPS52-86-009

NAVAL POSTGRADUATE SCHOOL

Monterey, California

EXPERIENCE WITH Ω Omega.

IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT, PART V 5.

Bruce J. MacLennan

February 1986

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

The work reported herein was supported by Contract from the Office of Naval Research.

This report was prepared by:

[illegible]

BRUCE J. MACLENNAN
Associate Professor

Reviewed by:

[illegible]

VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

[illegible]

KNEALE T. MARSHALL
Dean of Information and
Policy Science

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-86-009	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) EXPERIENCE WITH Ω IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT PART V		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(*s*) Bruce J. MacLennan		8. CONTRACT OR GRANT NUMBER(*s*)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943-5100		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N; RR015-08-01 N0001485WR24092
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, VA 22217		12. REPORT DATE February 1986 13. NUMBER OF PAGES 34
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*)		15. SECURITY CLASS. (*of this report*) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (*of this Report*) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)		

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

This is the fifth report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report presents a code generator operating on abstract syntax trees. The code generation process is implemented as an evaluator over a nonstandard domain. An implementation of the code generator is listed in the appendices.

EXPERIENCE WITH Ω

IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT PART V

Bruce J. MacLennan
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract:

This is the fifth report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report presents a code generator operating on abstract syntax trees. The code generation process is implemented as an evaluator over a nonstandard domain. An implementation of the code generator is listed in the appendices.

1. Introduction

Our goal in this series of reports* [MacLennan85b, MacLennan85c, MacLennan86a, MacLennan86b] is to explore in the context of a very simple language the use of the Ω programming notation [MacLennan83, MacLennan85a] to implement some of the tools that constitute a programming environment.

In this report we define a code generator for abstract programs. The code generator will be a member of the same family as the interpreter and the unparser. That is, it will be an evaluator for abstract programs defined on the domain of code sequences. First we discuss machine and run-time structure; next, informal translations; and finally present the translation rules.

2. Target Machine Structure

We will generate code for a stack machine with several special purpose registers (EP, SP) and several temporary registers (T1, T2). It has the following instructions:

* Support for this research was provided by the Office of Naval Research under contract N00014-86-WR-24092.

- LDC *k* – load constant
- ADD, SUB, MUL, DIV, EQL, etc. – arithmetic
- JMP *l*, JMPT *l* – unconditional jump, jump on true
- LBL *l* – define label
- SKIP *δ* – skip down static chain
- LOD – load contents of variable
- ENTER, EXIT – block control
- CALL, RETURN – function control
- PUSH *r*, POP r – stack control
- BREAK – enter debugger

3. Run-Time Structure

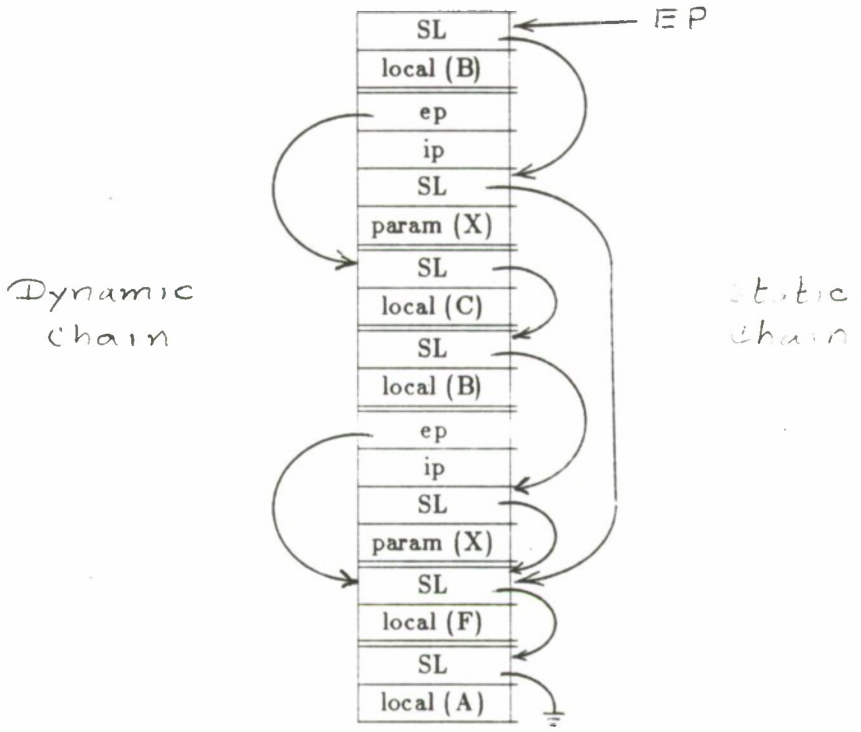
We use a conventional static-chain implementation for statically-scoped languages. Note that this stack-based activation record structure will not support function-valued functions, which are supported by the interpreter. This incompatibility between the interpreter and code generator is very serious, but not addressed in the present report, since it would not affect the use of Ω as a tool for writing the code generator. Exercise for the reader: define a non-stack-based activation record structure that solves this problem.

Consider the following program:

```
[let A = 1
 [func F X =
    [let B = (X x A)
     [let C = 3
      (if (X > 0)
       then F (C + (X - B))
       else 0 ) ] ]
```

```
F (A x 2) | |
```

This diagram illustrates the run-time data structures when execution is within the '**let** B = ...' block on the recursive invocation of F:



Notice how the static links for both of F's activation records point to the environment defining F. The ep/ip pair is the dynamic link.

4. Informal Translation Rules

4.1 Constants, Variables and Applications

For constants we merely stack the constant value:

$$k \implies \text{LDC } k$$

For variables we must first scan down the static chain to the environment of definition of the variable. Then the value of the local variable from that environment's activation record can be loaded onto the stack:

$$v \implies \begin{array}{l} \text{SKIP } \delta \\ \text{LOD} \end{array}$$

where δ is the static distance to v's activation record.

The code for an application is illustrated by this example:

$$X+Y \implies \begin{array}{l} X \\ Y \\ \text{ADD} \end{array}$$

The X and Y on the right represent the code corresponding to the X and Y on the left. Thus we generate code that executes X and Y in order and leaves their values on the stack, where they can be popped by ADD.

4.2 Conditional Expression

Code for a conditional first evaluates the condition, leaving a Boolean value on the stack. A JMPT instruction can then be used to test this value, skipping the alternate and jumping to the consequent when the value is **true**.

$$\begin{array}{l} (\textbf{if } B \\ \textbf{then } T \\ \textbf{else } F\,) \end{array} \implies \begin{array}{l} B \\ \text{JMPT } \tau \\ F \\ \text{JMP } \omega \\ \text{LBL } \tau \\ T \\ \text{LBL } \omega \end{array}$$

Of course the code for the alternate must end with a JMP to skip the consequent.

4.3 Blocks

The first step in the code for a block is the evaluation of the bound value E in the surrounding context. Two macroinstructions, ENTER and EXIT, surround the block body B, and handle the entry and exit of the block context:

$$[\textbf{let}\ v = E \quad B] \Longrightarrow \begin{array}{l} E \\ \text{ENTER} \\ B \\ \text{EXIT} \end{array}$$

The ENTER macroinstruction must create the block's activation record, incorporating the bound value, and link the activation record into the static chain. It is equivalent to the following operations:

```
PUSH EP    {SL}
EP← SP     {set EP}
```

That is, we push the old value of the EP register (which is a pointer to the surrounding context) onto the stack, thus forming the static link of the new activation record. The value of the bound value E is already on the stack, where it will be accessible as the local value in the new activation record. Transferring the contents of the stack pointer (SP) to the environment pointer (EP) installs the new activation record as the active one.

The EXIT macroinstruction must save the value computed by the block (which is on the top of the stack) while the block's activation record is deleted. Its code expansion is straight-forward:

```
POP T1     {block value}
POP EP     {SL}
POP –      {local}
PUSH T1    {block value}
```

4.4 Function Definition

Consider a function definition such as the following:

$$[\textbf{func}\ f\ n = B$$
$$X]$$

This is very much like a **let** block, except that execution of the function body B must be deferred until the function f is invoked:

```
JMP ω       {skip function body}
LBL φ       {entry point}
B           {body of function}
RETURN      {return to function}
LBL ω       {here to skip function body}
LDC φ       {stack entry point}
ENTER       {enter func. defn. block}
X           {body of func. defn. block}
EXIT        {exit func. defn. block}
```

The function body is represented by the LBL ϕ (which is its entry point), the code B, and the RETURN macroinstruction (which is discussed below). The JMP ω skips the function body, thus deferring its execution. The LDC ϕ stacks the entry point address as the local value of the function block, which is then ENTERed and EXITed in the usual way.

4.5 RETURN Instruction

The RETURN macroinstruction has the task of saving the function's value (which is on the top of the stack), restoring the caller's environment, deleting the function's activation record, leaving the function's value on the top of the stack, and resuming execution of the caller. The code to accomplish this is:

```
POP T1      {return value}
POP EP      {caller's EP}
POP T2      {caller's IP}
POP –       {SL}
POP –       {param}
PUSH T1     {return value}
JMP T2      {resume caller}
```

The first POP saves the function's value in temporary register T1. The second restores the callers environment from the dynamic link (EP/IP pair). The third saves the caller's resumption address in temporary register T2. The next two POPs delete the function's activation record. The PUSH instruction puts the function's value back on the top of the stack, and the indirect JMP through T2 transfers control back to the caller.

4.6 Function Invocation

The code sequence for the function application '$f\ X$' is as follows:

$$f\ X \Longrightarrow \begin{array}{l} X \\ \text{SKIP } \delta \\ \text{LOD} \\ \text{SKIP } \delta+1 \\ \text{CALL} \end{array}$$

where δ is static distance to f's environment of definition. The first SKIP moves to the activation record of the function block so that the LOD can access the entry address. The second SKIP, which goes one static link further, accesses the environment of definition of the function. The CALL macroinstruction completes the invocation process.

The CALL macroinstruction has the task of constructing an activation record for the callee and transferring control to the callee. This is accomplished by the following code expansion:

```
POP T1        {get env. of defn.}
POP T2        {get entry address}
PUSH T1       {static link}
PUSH ρ        {caller's IP}
PUSH EP       {caller's EP}
EP← SP−2      {callee's SL}
JMP T2        {enter function}
LBL ρ         {return location}
```

On entry to the CALL macroinstruction the top of the stack is the environment of definition of the callee, the second on the stack is the entry point address, and the third on the stack is the actual parameter value:

env. of defn.
entry point
actual
⋮

The first two are saved in registers T1 and T2. The actual parameter is left on the stack to form the first component of the callee's activation record. The next component is its static link (whose value was saved in register T1). Then we save the caller's IP (the resumption address ρ) and EP (which was in the EP register); together they constitute the dynamic link back to the caller. Finally, EP← SP−2 installs the callee's activation record as the active one, and the indirect JMP through T2 transfers control to the function. The LBL ρ of course defines the return point in the caller. (Exercise for the reader: Why '−2' in 'EP← SP−2'?) The completed activation record looks like this:

ep
ip
SL
param.
⋮

4.7 Example

Consider the following simple program:

```
[let K = 4
 [func fac n =
  (if (n = 0)
   then 1
   else (n x fac (n − 1)) )
  fac K ] ]
```

The following code will be generated:

```
LDC 4       local value K = 4
ENTER       enter 'let K = '
JMP L3      skip body of fac
LBL L4      entry point of fac
SKIP 0      access formal n
LOD         fetch value n
LDC 0       stack 0
EQL         compare, (n = 0)
JMPT L1     if true, skip alternate
SKIP 0      access formal n
LOD         fetch value n (to multiply)
SKIP 0      access formal n
LOD         fetch value n (to subtract)
LDC 1       stack 1
SUB         compute actual param (n - 1)
SKIP 1      access defn of fac
LOD         fetch entry point address
SKIP 2      access fac's env. of defn.
CALL        call fac (n - 1)
MUL         multiply n by result of fac
JMP L2      skip consequent of if
LBL L1      alternate of if:
LDC 1       stack 1
LBL L2      end of if
RETURN      return from fac
LBL L3      here to skip over fac
LDC L4      stack entry point of fac
ENTER       enter 'func fac = ' block
```

SKIP 1	access context of K
LOD	fetch value of K
SKIP 0	access context of fac
LOD	stack entry point of fac
SKIP 1	access env. of defn. of fac
CALL	call fac K
EXIT	exit '**func** fac = '
EXIT	exit '**let** K = '

Exercise for the reader: trace the execution of this program showing all stack states.

5. Code Generation

5.1 Introduction

The code generator is like Eval and Unparse, except that we change the domain on which the evaluation is done:

Unparse(E) $\Rightarrow$ "(3+ 5)"

Eval(E,C) $\Rightarrow$ 8

CodeGen(E,C) $\Rightarrow$ < LDC [3], LDC[5], ADD>

Notice that the "value" computed by CodeGen is a list of target machine instructions.

5.2 Code Generation Relations

The relations required for code generation are exact analogs of the Eval and Value relations in the interpreter:

- CodeGen (E, C)

  request code generation for E in context C

  Degree (CodeGen, 2), Domain (expr, 1, CodeGen), Domain (Context, 2, CodeGen).

- Code (U, E, C)

  U is the code for E in C

  Function (Code, expr×Context, code-list).

5.3 Constants

The code for a constant is simply the appropriate LDC instruction, which we assume to be generated by the function Con:

*CodeGen (E, C), Con (E), LitVal (V, E)

$\Rightarrow$ Code (< Con[V]> , E).

Note that because the range of Code is defined to be a code *list*, it is necessary to return Con[V] as a one-element list.

5.4 Applications

We need an additional relation, OpCode, which is a table giving the target machine opcode for each primitive operator. This relation corresponds to the Meaning relation of the interpreter and the Template relation of the unparser.

- OpCode (F, N)
- F is the opcode for N
- Function (OpCode, string, operation).

The analysis rule for applications must request the code generation of the two argument expressions:

*CodeGen (E, C). Appl (E), Left (X, E), Right (Y, E)

$\Longrightarrow$ CodeGen (X, C), CodeGen (Y, C).

The synthesis rule catenates the code sequence for the arguments with the appropriate arithmetic operation found in OpCode:

Appl (E), Op (N, E), Left (X, E), Right (Y, E),

*Code (U, X, C), *Code (V, Y, C), OpCode (F, N)

$\Longrightarrow$ Code (U ^ V ^ $<F>$, E, C).

Note that the opcode is made into a one element list so that it can be catenated with the code lists U and V.

5.5 Conditionals

The analysis rule for conditionals requests the code generation of the three parts of the conditional:

*CodeGen (E, C), ConEx (E), Cond (B, E), Conseq (T, E), Alt (F, E)

$\Longrightarrow$ CodeGen (B, C), CodeGen (T, C), CodeGen (F, C).

The synthesis rule assembles these with the appropriate jump instructions:

ConEx (E), Cond (B, E), Conseq (T, E), Alt (F, E)

*Code (U, B, C), *Code (V, T, C), *Code (W, F, C), *Avail (τ, ω)

$\Longrightarrow$ Code (

U ^

< JMPT [τ]> ^

W ^

< JMP [ω], LBL [τ]> ^

V ^

< LBL [ω]> , E, C).

The only complication is that unique lables τ and ω must be generated.

5.6 Block Structure

Contexts will be computed during code generation just as they are during evaluation. However, a name is bound to its ***static nesting level*** instead of its value (which is not known until runtime).

Variable lookup is requested by the Access relation: its static nesting level is returned in the Location relation.

- Access (N, D, E, C)

  access N in D for E in C

  Function (Access, expr$\times$Context, string$\times$Context).

- Location (L, E, C)

  L is the location for E in C

  Function (Location, expr$\times$Context, integer).

The rules governing the Access process are exact analogs of the Lookup rules in the interpreter:

*Access (N, D, E, C), Binds (D, N, L),

$\Longrightarrow$ Location (L, E, C)

else *Access (N, D, E, C), Nonlocal (D', D)

$\Longrightarrow$ Access (N, D', E, C)

else *Access (N, D, E, C)

$\Longrightarrow$ Break ("Undefined: " ^ N, E, C).

5.7 Variables

The analysis rule for variables simply request that Access determine the variable's location:

*CodeGen (E, C), Var (E), Ident (N, E)

$\Longrightarrow$ Access (N, C, E, C).

The synthesis rule waits for the static distance to be returned in Location, and incorporates it into the appropriate SKIP instruction:

Var (E), *Location (L, E, C), Binds $(C, -, K)$, ¬Rator $(E, -)$

$\Longrightarrow$ Code (< SKIP $[K-L]$, LOD> , E, C).

The condition '¬Rator $(E, -)$' is a bit of a kluge; it prevent the activation of this rule on variables that happen to be the operator of a function application, which must be handled differently. A runtime structure that supported function-valued functions (and variables) would eliminate the need for this kluge: exercise for the reader.

5.8 Blocks

The analysis rule for blocks requests code generation for the bound value and the block's body.

*CodeGen (E, C), Block (E), BndVar (N, E), BndVal (X, E), Body (B, E),

Binds $(C, -, K)$, *Avail (D)

$\Longrightarrow$ Context (D), Binds $(D, N, K+1)$, Nonlocal (C, D), CodeGen (X, C), CodeGen (B, D).

The bound value's code is generated at the same static nesting level as the block (K); the body is generated at a level one greater $(K+1)$. The synthesis rule merely catenates the code sequences with the

ENTER and EXIT instructions:

Block (E), BndVal (X, E), Body (B, E), *Code (U, X, C), *Code (V, B, D), Nonlocal (C, D)

$\Longrightarrow$ Code (U ^ < ENTER> ^ V ^ < EXIT> , E, C).

5.9 Function Definition

Code is generated for a function definition in very much the same way as for a block. The analysis rule requests code generation for the body of the function and the body of the function block, but this requires the creation of two new contexts:

*CodeGen (E, C), FunDef (E), FunName (F, E), FunFormal (N, E),

FunBody (B, E), FunScope (X, E), Binds (C, – , K), *Avail (D, A)

$\Longrightarrow$ Context (D), Nonlocal (C, D), Binds (D, F, K+1>), CodeGen (X, D),

Context (A), Nonlocal (D, A), Binds (A, N, K+2), CodeGen (B, A)

The context D represents the context of the function definition block, which binds F to static nesting level $K+1$ (i.e., one more than that of the surrounding context). Code for the body X of the function definition block is generated in this context D. The context A represents the context of the function's body, which binds the formal A to its static nesting level ($K+2$, i.e., one more than D's). A is the context in which code is generated for the function's body; notice that the nonlocal environment of A includes D, thus permitting recursive function invocations.

The synthesis rule gathers the code generated for the function and block bodies, and assembles it into the complete code sequence:

FunDef (E), FunBody (B, E), FunScope (X, E), Nonlocal (C, D),

*Code (U, B, A), *Code (V, X, D), *Avail (ω, ϕ)

$\Longrightarrow$ Code (

< JMP [ω], LBL [ϕ]> ^ U ^

< RETURN, LBL [ω],

LDC [ϕ], ENTER> ^ V ^

< EXIT> , E, C).

The label ϕ is the function's entry point (which is left on the stack); the label ω is for skipping over the function's body, so it will not be executed until it is called.

6. Function Invocation

For function invocations the analysis rule requests code generation for the actual parameter, and lookup for the function's name:

*CodeGen (E, C), Call (E), Rator (F, E), Rand (X, E), Var (F), Ident (N, F)

$\Rightarrow$ Access (N, C, F, C), CodeGen (X, C).

Note that the code generator requires the Rator to be a variable, and also interprets that variable as the function's name (as opposed to a variable pointing to the function, etc.).

The synthesis rules picks up from Location the static nesting level at which the function was defined, and uses it to assemble the code sequence:

Call (E), Rator (F, E), Rand (X, E), *Location (L, F, C), *Code (V, X, C), Binds $(C, -, K)$

$\Rightarrow$ Code (V ^< SKIP $[K-L]$, LOD, SKIP $[K-L+1]$, CALL> . E, C).

The first SKIP accesses the context in which the function was defined, since the local value of this context is the entry point address of the function; see 5.9 Function Definition above. The LOD moves the entry point address to the top of the stack. The second SKIP goes one further than the previous, which accesses the environment of definition of the function. The actual parameter, entry point address and environment of definition are left on the stack for the CALL macroinstruction (see 4.6, Function Invocation).

7. References

[MacLennan83] MacLennan, B. J., A View of Object-Oriented Programming, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-001, February 1983.

[MacLennan84] MacLennan, B. J., The Four Forms of Ω: Alternate Syntactic Forms for an Object-Oriented Language, Naval Postgraduate School Computer Science Department Technical Report NPS52-84-026, December 1984.

[MacLennan85a] MacLennan, B. J., A Simple Software Environment Based on Objects and Relations, *Proc. of ACM SIGPLAN 85 Conf. on Language Issues in Prog. Environments,* June 25-28, 1985, and Naval Postgraduate School Computer Science Department Technical Report NPS52-85-005, April 1985.

[MacLennan85b] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part I, Naval Postgraduate School Computer Science Department Technical Report NPS52-85-006, May 1985.

[MacLennan85c] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part II, Naval Postgraduate School Computer Science Department Technical Report NPS52-85-015, December 1985.

[MacLennan86a] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part III, Naval Postgraduate School Computer Science Department Technical Report NPS52-86-004, January 1986.

[MacLennan86b] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part IV, Naval Postgraduate School Computer Science Department Technical Report NPS52-86-007, January 1986.

[McArthur84] McArthur, Heinz M., *Design and Implementation of an Object-Oriented, Production-Rule Interpreter,* MS Thesis, Naval Postgraduate School Computer Science Department, December 1984.

[Ufford85] Ufford, Robert P., *The Design and Analysis of a Stylized Natural Grammar for an Object Oriented Language (Omega),* MS Thesis, Naval Postgraduate School Computer Science Department, June 1985.

APPENDIX A: Prototype Programming Environment

The following is a loadable input file for the code generator described in this report.. Its operation requires the PI-4 system listed in Part IV [MacLennan86b], which is not reproduced here. The complete system is accepted by the McArthur interpreter [McArthur84], which differs in a few details from the Ω notation used in this report (see [MacLennan84]). A transcript of a test execution of this environment is shown in Appendix B.

```
!____________________________________________
!
! CODE GENERATOR
!
!____________________________________________

! Reducing Append Function

fn ap[LL]:
  if LL= Nil -> []
  else append [first [LL], ap [rest [LL]]];

! Relations

newrelation {"CodeGen"};
newrelation {"Code"};
newrelation {"Access"};
newrelation {"Location"};
newrelation {"OpCode"};
newrelation {"CreateConExCode"};
newrelation {"CreateBlockCode"};
newrelation {"CreateFunDefCode"};
newrelation {"CreateFunDef2Code"};
newrelation {"newlab"};

newrelation {"LastLabel"}.

! Machine Op Codes

! Alias procedure to define niladic opcodes:

newrelation {"alias"}.

act {< < if *alias (A, s) -> define {root, s, s}, A (s); > > }.

alias {"LOD"};

alias {"ENTER"};

alias {"EXIT"};

alias {"CALL"};

alias {"RETURN"};

alias {"BREAK"};

! Monadic opcodes:

fn LDC [k] : "LDC " + k;

fn JMP [l] : "JMP " + l;

fn JMPT [l] : "JMPT " + l;

fn LBL [l] : "LBL " + l;

fn SKIP [delta] : "SKIP " + int_str [delta];

fn PUSH [r] : "PUSH " + r;

fn POP [r] : "POP " + r;

! Opcodes used in operator applications:

OpCode ("ADD", "+"),

OpCode ("SUB", "-"),

OpCode ("MUL", "x"),
OpCode ("DIV", "/"),

OpCode ("EQL", "= "),

OpCode ("GTR", "> ").
```

```
! CODE GENERATOR RULES

define {root, "CodeGenRules", < <

! Incomplete Programs

if *CodeGen (E, C), Undef (E)

-> Code ([BREAK], E, C);

! Constants

if *CodeGen (E, C), Con (E), Litval (V, E)

-> Code ([LDC [int_str [V]]], E, C);

! Applications: Analysis

if *CodeGen (E, C), Appl (E), Left (X, E), Right (Y, E)

-> CodeGen (X, C), CodeGen (Y, C);

! Applications: Synthesis

if Appl (E), Op (N, E), Left (X, E), Right (Y, E), *Code (U, X, C), *Code (V, Y, C), OpCode (F, N)

-> Code (ap [[U, V, [F]]], E, C);

! Conditionals: Analysis

if *CodeGen (E, C), ConEx (E), Cond (B, E), Conseq (T, E), Alt (F, E)

-> CodeGen (B, C), CodeGen (T, C), CodeGen (F, C);

! Conditionals: Synthesis

if ConEx (E), Cond (B, E), Conseq (T, E), Alt (F, E),

 *Code (U, B, C), *Code (V, T, C), *Code (W, F, C)

-> CreateConExCode (U, V, W, E, C, newlab {}, newlab {});
```

```
if *CreateConExCode (U, V, W, E, C, tau, omega)
-> Code (ap [[
 U,
 [JMPT [tau]],
 W,
 [JMP [omega], LBL [tau]],
 V,
 [LBL [omega]]]], E, C);

! Name Lookup Rules

if *Access (N, D, E, C), Binds (D, N, L)
-> Location (L, E, C)

else if *Access (N, D, E, C), Nonlocal (Dprime, D)
-> Access (N, Dprime, E, C)

else if *Access (N, D, E, C)
-> Break ("Undefined: " + N, E, C);

! Variables: Analysis

if *CodeGen (E, C), Var (E), Ident (N, E)
-> Access (N, C, E, C);

! Variables: Synthesis

if Var (E), *Location (L, E, C), Binds (C, - , K), ~Rator (E, - )
-> Code ([SKIP [K-L], LOD], E, C);

! Blocks: Analysis

if *CodeGen (E, C), Block (E), BndVar (N, E), BndVal (X, E), Body (B, E), Binds (C, - , K)
```

```
-> CreateBlockCode (C, N, K, X, B, newobj {});

if *CreateBlockCode (C, N, K, X, B, D)
-> Context (D), Binds (D, N, K+1), Nonlocal (C, D), CodeGen (X, C), CodeGen (B, D);

! Blocks: Synthesis

if Block (E), BndVal (X, E), Body (B, E), *Code (U, X, C), *Code (V, B, D), Nonlocal (C, D)
-> Code (ap [[U, [ENTER], V, [EXIT]]], E, C);

! Function Definition: Analysis

if *CodeGen (E, C), FunDef (E), FunName (F, E), FunFormal (N, E), FunBody (B, E), FunScope (X, E
 Binds (C, - , K)
-> CreateFunDefCode (C, F, K, X, N, B, E, newobj {}, newobj {}, newlab {});

if *CreateFunDefCode (C, F, K, X, N, B, E, A, D, phi)
-> Context (D), Nonlocal (C, D), Binds (D, F, K+1), CodeGen (X, D), Context (A), Nonlocal (D, A),
 Binds (A, N, K+2), CodeGen (B, A);

! Function Definition: Synthesis

if FunDef (E), FunBody (B, E), FunScope (X, E), *Code (U, B, A), *Code (V, X, D), Nonlocal (C, D)
-> CreateFunDef2Code (newlab {}, newlab {}, U, V, E, C);

if *CreateFunDef2Code (omega, phi, U, V, E, C)
-> Code (ap [[
 [JMP [omega], LBL [phi]], U,
 [RETURN, LBL [omega],
 LDC [phi], ENTER], V,
 [EXIT]]], E, C);

! Function Invocation: Analysis
if *CodeGen (E, C), Call (E), Rator (F, E), Rand (X, E), Var (F), Ident (N, F)

-> Access (N, C, F, C), CodeGen (X, C);


! Function Invocation: Synthesis


if Call (E), Rator (F, E), Rand (X, E), *Location (L, F, C), *Code (V, X, C), Binds (C, - , K)

-> Code (ap [[V, [SKIP [K-L], LOD, SKIP [K-L+ 1], CALL]]], E, C);


! New Label Generator


if *newlab (A), *LastLabel (n)

-> A ("L" + int_str [n]), LastLabel (n + 1);


> > }.


act {CodeGenRules}.
```

```
! Code Generator Commands

newrelation {"CodeGenPending"}.

define {root, "CodeGenComRules", < <

! codegen Command

if *Command ("codegen"), CurrentNode (E), CurrentContext (C)
-> CodeGen (E, C), CodeGenPending (E), CommandPending (E);

if Code (V, E, C), *CodeGenPending (E), *CommandPending (- )
-> displayn {"Code generation completed."};

! showcode Command

if *Command ("showcode"), CurrentNode (E), Code (V, E, C)
-> displayn {V};

if *Command ("showcode"), CurrentNode (E), ~Code (V, E, C)
-> displayn {"No code available"};
> > }.

act {CodeGenComRules}.

define {root, "CodeGenTests", < <

if *Test (A, 10) -> { Script {[
    "begin", "let", "K", "#", 4, "next", "func", "fac", "n",
    "if", "= ", "var", "n", "next", "#", 0, "out", "next", "#", 1, "next",
    "x", "var", "n", "next", "call", "var", "fac", "next",
    "-", "var", "n", "next", "#", 1, "root", "in", "next", "in", "next",
    "call", "var", "fac", "next", "var", "K", "root", "codegen", "showcode"
    ]};
```

```
  A ("Test done");

  };

> > }.

act {CodeGenTests}.

LastLabel (0);

if *CurrentContext (— ) -> CurrentContext (newobj {}).

if CurrentContext (C) -> Binds (C, "", 0).

displayn {"PI-5 System Loaded."}.
```

APPENDIX B: Transcript of Ω Session

The following is a transcript of an Ω session illustrating the operation of the prototype programming environment shown in Appendix A. The assertion 'Script {testscript}' causes the commands in testscript to be executed in order. The *n*th testscript is executed by 'Test{*n*}'. Each command is printed on a separate line, followed by whatever output is generated by the programming environment. This transcript was produced by the McArthur interpreter [McArthur84].

```
% omega

OMEGA-1   11/30/84

Use Cntl-D or exit{} to quit.

For help, enter help{"?"}.

To report a bug, enter Bugs{}.

newrelation rule activated.

> do{"PI4.rul"}. do{"PI5.rul"}.

PI-4 System loaded

OK

> PI-5 System Loaded.

OK

> Test{10}.

 ... begin

 ... K let

< expr>

 ... 4 #

 ... next

< expr>

 ... fac n func

 ... if
```

```
< expr>
 ... =
< expr>
 ... n var
 ... next
< expr>
 ... 0 #
 ... out
(n = 0)
 ... next
< expr>
 ... 1 #
 ... next
< expr>
 ... x
< expr>
 ... n var
 ... next
< expr>
 ... call
 ... fac var
 ... next
< expr>
 ... -
< expr>
 ... n var
 ... next
< expr>
```

```
 ... 1 #

 ... root

let  K =  4

  func fac n =

    (if (n =  0)

     then 1

     else (n x fac (n - 1)) )

    < expr>

 ... in

4

 ... next

func fac n =

  (if (n =  0)

   then 1

   else (n x fac (n - 1)) )

  < expr>

 ... in

(if (n =  0)

 then 1

 else (n x fac (n - 1)) )

 ... next

< expr>

 ... call

 ... fac var

 ... next
```

```
< expr>

 ... K var

 ... root

[let  K =  4

  [func fac n =

    (if (n =  0)

     then 1

     else (n x fac (n - 1)) )

    fac K ] ]

 ... codegen

Code generation completed.

 ... showcode

[LDC 4, ENTER, JMP L3, LBL L4, SKIP 0, LOD, LDC 0, EQL, JMPT L1, SKIP 0, LOD,

 SKIP 0, LOD, LDC 1, SUB, SKIP 1, LOD, SKIP 2, CALL, MUL, JMP L2, LBL L1, LDC 1,

 LBL L2. RETURN, LBL L3, LDC L4, ENTER, SKIP 1, LOD, SKIP 0, LOD, SKIP 1, CALL,

 EXIT. EXIT]

>  exit{}.

Goodbye.

%
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 2

Dudley Knox Library
Code 0142
Naval Postgraduate School
Monterey, CA 93943 2

Office of Research Administration
Code 012A
Naval Postgraduate School
Monterey, CA 93943 1

Chairman, Code 52
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 40

Associate Professor Bruce J. MacLennan
Code 52ML
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 12

Dr. Robert Grafton
Code 433
Office of Naval Research
800 N. Quincy
Arlington, VA 22217-5000 1

Dr. David Mizell
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106 1

Dr. Stephen Squires
DARPA
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209 1

Professor Jack M. Wozencraft, 62Wz
Department of Electrical and Comp. Engr.
Naval Postgraduate School
Monterey, CA 93943 1

Professor Rudolf Bayer
Institüt für Informatik
Technische Universität
Postfach 202420
D-8000 Munchen 2
West Germany 1

Dr. Robert M. Balzer
USC Information Sciences Inst.
4676 Admiralty Way
Suite 10001
Marina del Rey, CA 90291 1

Mr. Ronald E. Joy
Honeywell, Inc.
Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, MI 55402 1

Mr. Ron Laborde
INMOS
Whitefriars
Lewins Mead
Bristol
Great Britain 1

Mr. Lynwood Sutton
Code 424, Building 600
Naval Ocean Systems Center
San Diego, CA 92152 1

Mr. Jeffrey Dean
Advanced Information and Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040 1

Mr. Jack Fried
Mail Station D01/31T
Grumman Aerospace Corporation
Bethpage, NY 11714 1

Mr. Dennis Hall
New York Videotext
104 Fifth Avenue, Second Floor
New York, NY 10011 1

Professor S. Ceri
Laboratorio di Calcolatori
Departimento di Elettronica
Politecnico di Milano
20133 - Milano
Italy 1

Mr. A. Dain Samples
Computer Science Division - EECS
University of California at Berkeley
Berkeley, CA 94720 1

Antonio Corradi
Dipartimento di Elettronica
Informatica e Sistemistica
Universita Degli Studi di Bologna
Viale Risorgimento, 2

Bologna
Italy 1

Dr. Peter J. Welcher
Mathematics Dept., Stop 9E
U.S. Naval Academy
Annapolis, MD 21402 1

Dr. John Goodenough
Wang Institute
Tyng Road
Tyngsboro, MA 01879 1

Professor Richard N. Taylor
Computer Science Department
University of California at Irvine
Irvine, CA 92717 1

Dr. Mayer Schwartz
Computer Research Laboratory
MS 50-662
Tektronix, Inc.
Post Office Box 500
Beaverton, OR 97077 1

Professor Lori A. Clarke
Computer and Information Sciences Department
LGRES ROOM A305
University of Massachusetts
Amherst, MA 01003 1

Professor Peter Henderson
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794 1

Dr. Mark Moriconi
SRI International
333 Ravenswood Avenue
Manlo Park, CA 95025 1

Professor William Waite
Department of Electrical and Computer Engineering
The University of Colorado
Campus Box 425
Boulder, CO 80309-0425 1

Professor Mary Shaw
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213 1

Dr. Warren Teitelman
Engineering/Software
Sun Microsystems Federal, Inc.
2550 Garcia Avenue

Mountain View, CA 94031	1
Prof. Raghu Ramakrishnan Univ. of Texas at Austin Dept. of Computer Science Austin, TX 79712	1